UNITED STATES PATENT APPLICATION

FOR

A METHOD FOR PROVIDING EXTENDED PRECISION IN SIMD VECTOR ARITHMETIC OPERATIONS

Inventors:

Timothy van Hook
Peter Hsu
William Huffman
Henry Moreton
Earl Killian

Prepared by:
WAGNER, MURABITO & HAO
Two North Market Street
Third Floor
San Jose, California 95113

SGI-15-4-458.00

October 6, 1997

Ins a1

A METHOD FOR PROVIDING EXTENDED PRECISION IN SIMD VECTOR ARITHMETIC OPERATIONS

FIELD OF THE INVENTION

The present claimed invention relates to the field of single instruction multiple data (SIMD) vector process. More particularly, the present claimed invention relates to extended precision in SIMD vector arithmetic operations.

BACKGROUND ART

Today, most processors in computer systems provide a 64-bit datapath architecture. The 64-bit datapath allows operations such as read, write, add, subtract, and multiply on the entire 64 bits of data at a time. This added bandwidth has significantly improved performance of the processors.

However, the data types of many real world applications do not utilize the full 64 bits in data processing. For example, in digital signal processing (DSP) applications involving audio, video, and graphics data processing, the light and sound values are usually represented by data types of 8, 12, 16, or 24 bit numbers. This is because people typically are not able to distinguish the levels of light and sound beyond the levels represented by these numbers of bits. Hence, DSP applications typically require data types far less than the full 64 bits provided in the datapath in most computer systems.

In initial applications, the entire datapath was used to compute an image or sound values. For example, an 8 or 16 bit number representing a pixel or sound value was loaded into a 64-bit number. Then, an arithmetic operation, such as an add or multiply, was performed on the entire 64-bit number. This method proved inefficient however, as it was soon realized that not all the data bits were being utilized in the process since digital representation of a sound or pixel requires far fewer bits. Thus, in order to utilize the entire datapath, a multitude of smaller numbers were packed into the 64 bit doubleword.

Furthermore, much of data processing in DSP applications involve repetitive and parallel processing of small integer data types using loops. To take advantage of this repetitive and parallel data process, a number of today's processors implements single instruction multiple data (SIMD) in the instruction architecture. For instance, the Intel Pentium MMX^{TM} chips incorporate a set of SIMD instructions to boost multimedia performance.

Prior Art Figure 1 illustrates an exemplary single instruction multiple data instruction process. Exemplary registers, vs and vt, in a processor are of 64-bit width. Each register is packed with four 16-bit data elements fetched from memory: register vs contains vs[0], vs[1], vs[2], and vs[3] and register vt contains vt[0], vt[1], vt[2], and vt[3]. The registers in essence contain a vector of N elements. To add elements of matching index, an add instruction adds, independently, each of the element pairs of matching index from vs and vt. A third register, vd, of 64-bit width may be used to store the result. For example, vs[0] is added to vt[0] and its result is stored into vd[0]. Similarly, vd[1], vd[2], and vd[3] store the sum of vs and vd elements of corresponding indexes. Hence, a single add operation on the 64-bit vector results in 4 simultaneous additions on each of the 16-bit elements. On the other hand, if 8-bit elements were packed into the registers, one add operation performs 8 independent additions in parallel. Consequently, when a SIMD arithmetic instruction, such as addition, subtraction, or multiply, is performed on the data in the 64-bit datapath, the operation actually performs multiple numbers of operations independently and in parallel on each of the smaller elements comprising the 64 bit datapath.

Unfortunately however, an arithmetic operation such as add and multiply on SIMD vectors typically increases the number of significant bits in the result. For instance, an addition of two n-bit numbers may result in a number of n+1 bits. Moreover, a multiplication of two n-bit numbers produces a number of 2n bit width. Hence, the results of an arithmetic operation on a SIMD vector may not be accurate to a desired significant bit.

Furthermore, the nature of multimedia DSP applications often increases inaccuracies in significant bits. For example, many DSP algorithms implemented in DSP applications require a series of computations producing partial results that are larger or bigger, in terms of significant number of bits, than the final result. Since the final result does not fully account for the significant bits of these partial results, the final result may not accurately reflect the ideal result, which takes into account all significant bits of the intermediate results.

To recapture the full significant bits in a SIMD vector arithmetic operation, the size of the data in bits for each individual element was typically boosted or promoted to twice the size of the original data in bits. Thus, for multiplication on 8-bit elements in a SIMD vector for instance, the 8-bit elements were converted (i.e., unpacked) into 16-bit elements containing 8 significant bits to provide enough space to hold the subsequent product.

Unfortunately however, the boost in the number of data bits largely undermined the benefits of SIMD vector scheme by reducing the speed of an arithmetic operation in half. This is because the boosting of data bits to twice the original size results in half as many data elements in a register. Hence, an operation on the entire 64-bit datapath comprised of 16-bit elements accomplishes only 4 operations in comparison to 8 operations on a 64-bit datapath comprised of 8-bit elements. In short, boosting a data size by X-fold results in performance reduction of (1/X)*100 percent. As a result, instead of an effective 64-bit datapath, the effective datapath was only 32-bits wide.

Thus, what is needed is a method and system for providing extended precision in SIMD vector arithmetic operations without sacrificing speed and performance.

SUMMARY OF THE INVENTION

The present invention provides extended precision in SIMD arithmetic operations in a processor having a register file and an accumulator. The register file is comprised of a plurality of general purpose registers of N bit width. The size of the accumulator is preferably an integer multiple of the size of the general purpose registers. The preferred embodiment uses registers of 64 bits and an accumulator of 192 bits. The present invention first loads, from a memory, a first set of data elements into a first vector register and a second set of data elements into a second vector register. Each data element comprises N bits. Next, an arithmetic instruction is fetched from memory and is decoded. Then, a first vector register and a second vector register are read from the register file as specified in the arithmetic instruction. The present invention then executes the arithmetic instruction on corresponding data elements in the first and second vector registers. The result of the execution is then written into the accumulator. Then, each element in the accumulator is transformed into an N-bit width element and written into a third register for further operation or storage in the memory.

In an alternative embodiment, the accumulator contains a third set of data elements. After the arithmetic operation between the data elements in the first and second vector registers, the result of the execution is added to the corresponding elements in the accumulator. The result of the addition is then written into the accumulator. Thereafter, each element in the accumulator is transformed into an N-bit width element and written into a third register for further operation or storage in the memory.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and form a part of this specification, illustrate embodiments of the invention and, together with the description, serve to explain the principles of the invention:

Prior Art Figure 1 illustrates an exemplary single instruction multiple data (SIMD) instruction method.

Figure 2 illustrates an exemplary computer system of the present invention.

Figure 3 illustrates a block diagram of an exemplary datapath including a SIMD vector unit (VU), a register file, and a vector load/store unit according to one embodiment of the present invention.

Figure 4 illustrates a more detailed datapath architecture including the accumulator in accordance with the present invention.

Figure 5 illustrates a flow diagram of general operation of an exemplary arithmetic instruction according to a preferred embodiment of the present invention.

Figure 6 illustrates element select format for 4 16-bit elements in a 64-bit register.

Figure 7 illustrates element select format for 8 8-bit elements in a 64-bit register.

Figure 8 illustrates an exemplary ADDA.fmt arithmetic operation between elements of exemplary operand registers vs and vt.

Figure 9 illustrates an exemplary ADDL.fmt arithmetic operation between elements of exemplary operand registers vs and vt.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

In the following detailed description of the present invention, numerous specific details are set forth in order to provide a thorough understanding of the present invention. However, it will be obvious to one skilled in the art that the present invention may be practiced without these specific details. In other instances well known methods, procedures, components, and circuits have not been described in detail so as not to unnecessarily obscure aspects of the present invention.

The present invention features a method for providing extended precision in single-instruction multiple-data (SIMD) arithmetic operations in a computer system. The preferred embodiment of the present invention performs integer SIMD vector arithmetic operations in a processor having 64-bit wide datapath within an exemplary computer system described above. Extended precision in the SIMD arithmetic operations are supplied through the use of an accumulator register having a preferred width of 3 times the general purpose register width. Although a datapath of 64-bits is exemplified herein, the present invention is readily adaptable to datapaths of other variations in width.

COMPUTER SYSTEM ENVIRONMENT

Figure 2 illustrates an exemplary computer system 212 comprised of a system bus 200 for communicating information, one or more central processors 201 coupled with the bus 200 for processing information and instructions, a computer readable volatile memory unit 202 (e.g., random access memory, static RAM, dynamic RAM, etc.) coupled with the bus 200 for storing information and instructions for the central processor(s) 201, a computer readable non-volatile memory unit (e.g., read only memory, programmable ROM, flash memory, EPROM, EEPROM, etc.) coupled with the bus 200 for storing static information and instructions for the processor(s).

Computer system 212 of Figure 2 also includes a mass storage computer readable data storage device 204 (hard drive, floppy, CD-ROM, optical drive, etc.) such as a magnetic or optical disk and disk drive coupled with the bus 200 for storing information and instructions. Optionally, system 212 can include a display device 205 coupled to the bus 200 for displaying information to the user, an alphanumeric input device 206 including alphanumeric and function keys coupled to the bus 200 for communicating information and command selections to the central processor(s) 201, a cursor control device 207 coupled to the bus for communicating user input information and command selections to the central processor(s) 201, and a signal generating device 208 coupled to the bus 200 for communicating command selections to the processor(s) 201.

According to a preferred embodiment of the present invention, the processor(s) 201 is a SIMD vector unit which can function as a coprocessor for a host processor (not shown). The VU performs arithmetic and logical operations on individual data elements within a data word using the instruction methods described below. Data words are treated as vectors of Nx1 elements, where N can be 8, 16, 32, 64, or multiples thereof. For example, a set of Nx1 data elements of either 8- or 16-bit fields comprises a data doubleword of 64-bit width. Hence, a 64 bit wide double word contains either 4 16-bit elements or 8 8-bit elements.

Figure 3 illustrates a block diagram of an exemplary datapath 300 including a SIMD vector unit (VU) 302, a register file 304, a vector load/store unit 318, and crossbar circuits 314 and 316 according to one embodiment of the present invention. The VU 302 executes an operation specified in the instruction on each element within a vector in parallel. The VU 302 can operate on data that is the full width of the local on-chip memories, up to 64 bits. This allows parallel operations on 8 8-bit, 4 16-bit, 2 32-bit, or 1 64-bit elements in one cycle. The VU 302 includes an accumulator 312 to hold values to be accumulated or accumulated results.

The vector register file is comprised of 32 64-bit general purpose registers 306 through 310. The general purpose registers 306 through 310 are visible to the programmer and can be used to store intermediate results. The preferred embodiment of the present invention uses the floating point registers (FGR) of a floating point unit (FPU) as its vector registers.

In this shared arrangement, data is moved between the vector register file 304 and memory with Floating Point load and store doubleword instructions through the vector load/store unit 318. These load and store operations are unformatted. That is, no format conversions are performed and therefore no floating-point exceptions can occur due to these operations. Similarly, data is moved between the vector register file 304 and the VU 302 without format conversions, and thus no floating-point exception occurs.

Within each register, data may be written, or read, as bytes (8-bits), short-words (16-bits), words (32-bits), or double-words (64-bits). Specifically, the vector registers of the present invention are interpreted in the following new data formats: Quad Half (QH), Oct Byte (OB), Bi Word (BW), and Long (L). In QH format, a vector register is interpreted as having 16-bit elements. For example, a 64-bit vector register is interpreted as a vector of 4 signed 16-bit integers. OB format interprets a vector register as being comprised of 8-bit elements. Hence, an exemplary 64-bit vector register is seen as a vector of 8 unsigned 8-bit integers. In BW format, a vector register is interpreted as having 32-bit elements. L format interprets a vector register as having 64-bit elements. These data types are provided to be adaptable to various register sizes of a processor. As described above, data format conversion is not necessary between these formats and floating-point format.

With reference to Figure 3, the present invention utilizes crossbar circuits to select and route elements of a vector operand. For example, the crossbar circuit 314 allows selection of elements of a given data type and pass on the selected elements as operands to VU 302. The VU 302 performs arithmetic operations on operands comprised of elements and outputs the result to another crossbar circuit 316. This crossbar circuit 316 routes the resulting elements to corresponding element fields in registers such as vd 310 and accumulator 312. Those skilled in the art will no doubt recognize that crossbar circuits are routinely used to select and route the elements of a vector operand.

With reference to Figure 3, the present invention also provides a special register, accumulator 312, of preferably 192-bit width. This register is used to store intermediate add, subtract, or multiply results generated by one instruction with the intermediate add, subtract, or multiply results generated by either previous or subsequent instructions. The accumulator 312 can also be loaded with a vector of elements from memory through a register. In addition, the accumulator 312 is capable for forwarding data to the VU 302, which executes arithmetic instructions. Although the accumulator 312 is shown to be included in the VU 302, those skilled in the art will recognize that it can also be placed in other parts of the datapath so as to hold either accumulated results or values to be accumulated.

Figure 4 illustrates a more detailed datapath architecture including the accumulator 312. In this datapath, the contents of two registers, vs and vt, are operated on by an ALU 402 to produce a result. The result from the ALU can be supplied as an operand to another ALU such as an adder/subtractor 404. In this datapath configuration, the accumulator 312 can forward its content to be used as the other operand to the adder/subtractor 404. In this manner, the accumulator 312 can be used as both a source and a destination in consecutive cycles without causing pipe stalls or data hazards. By thus accumulating the intermediate results in its expanded form in tandem with its ability to be used as both a source and a destination, the accumulator 312 is used to provide extended precision for SIMD arithmetic operations.

An exemplary accumulator of the present invention is larger in size than general purpose registers. The preferred embodiment uses 192-bit accumulator and 64-bit registers. The format of the accumulator is determined by the format of the elements accumulated. That is, the data types of an accumulator matches the data type of operands specified in an instruction. For example, if the operand register is in QH format, the accumulator is interpreted to contain 4 48-bit elements. In OB format, the accumulator is seen as having 8 24-bit elements. In addition, accumulator elements are always signed. Elements are stored from or loaded into the accumulator indirectly to and from the main memory by staging the elements through the shared Floating Point register file.

Figure 5 illustrates a flow diagram of an exemplary arithmetic operation according to a preferred embodiment of the invention. In step 502, an arithmetic instruction is fetched from memory into an instruction register. Then in step 504, the instruction is decoded to determine the specific arithmetic operation, operand registers, selection of elements in operand registers, and data types. The instruction opcode specifies an arithmetic operation such as add, multiply, or subtract in its opcode field. The instruction also specifies the data type of elements, which determines the width in bits and number of elements involved in the arithmetic operation. For example, OB data type format instructs the processor to interpret a vector register as containing 8 8-bit elements. On the other hand, QH format directs the processor to interpret the vector register as having 4 16-bit elements.

The instruction further specifies two operand registers, a first register (vs) and a second register (vt). The instruction selects the elements of the second register, vt, to be used with each element of the accumulator, and/or the first register, vs. For example, the present invention allows selection of one element from the second register to be used in an arithmetic operation with all the elements in the first register independently and in parallel. The selected element is replicated for every element in the first register. In the alternative, the present invention provides selection of all elements from the second register to be used in the arithmetic operation with all the elements in the first register. The arithmetic operation operates on the corresponding elements of the registers independently and in parallel. The present invention also provides an immediate value (i.e., a constant) in a vector field in the instruction. The immediate value is replicated for every element of the second register before an arithmetic operation is performed between the first and second registers.

According to the decoded instruction, the first register and the second register with the selected elements are read for execution of the arithmetic operation in step 506. Then in step 508, the arithmetic operation encoded in the instruction is executed using each pair of the corresponding elements of first register and the second register as operands. The resulting elements of the execution are written into corresponding elements in the accumulator in step 510. According to another embodiment of the present invention, the resulting elements of the execution are added to the existing values in the accumulator elements. That is, the accumulator "accumulates" (i.e., adds) the resulting elements onto its existing elements. The elements in the accumulator are then transformed into N-bit width in step 512. Finally, in step 514, the transformed elements are stored into memory. The process then terminates in step 516.

The SIMD vector instructions according to the present invention either write all 192 bits of the accumulator or all 64 bits of an FPR, or the condition codes. Results are not stored to multiple destinations, including the condition codes.

Integer vector operations that write to the FPRs clamp the values being written to the target's representable range. That is, the elements are saturated for overflows and under flows. For overflows, the values are clamped to the largest representable value. For underflows, the values are clamped to the smallest representable value.

On the other hand, integer vector operations that write to an accumulator do not clamp their values before writing, but allow underflows and overflows to wrap around the accumulator's representable range. Hence, the significant bits that otherwise would be lost are stored into the extra bits provided in the accumulator. These extra bits in the accumulator thus ensure that unwanted overflows and underflows do not occur when writing to the accumulator or FPRs.

SELECTION OF VECTOR ELEMENTS

The preferred embodiment of the present invention utilizes an accumulator register and a set of vector registers in performing precision arithmetic operations. First, an exemplary vector register, vs, is used to hold a set of vector elements. A second exemplary vector register, vt, holds a selected set of vector elements for performing operations in conjunction with the elements in vector register, vs. The present invention allows an arithmetic instruction to select elements in vector register vt for operation with corresponding elements in other vector registers through the use of a well known crossbar method. A third exemplary vector register, vd, may be used to hold the results of operations on the elements of the registers described above. Although these registers (vs, vt, and vd) are used to associate vector registers with a set of vector elements, other vector registers are equally suitable for present invention.

To perform arithmetic operations on desired elements of a vector, the present invention uses a well known crossbar method adapted to select an element of the vector register, vt, and replicate the element in all other element fields of the vector. That is, an element of vt is propagated to all other elements in the vector to be used with each of the elements of the other vector operand. Alternatively, all the elements of the vector, vt, may be selected without modification. Another selection method allows an instruction to specify as an element an immediate value in the instruction opcode vector field corresponding to vt and replicate the element for all other elements of vector vt. These elements thus selected are then passed onto the VU for arithmetic operation.

Figure 6 illustrates element select format for 4 16-bit elements in a 64-bit register. The exemplary vector register vt 600 is initially loaded with four elements: A, B, C, and D. The present invention allows an instruction to select or specify any one of the element formats as indicated by rows 602 through 610. For example, element B for vt 600 may be selected and replicated for all 4 elements as shown in row 604. On the other hand the vt 600 may be passed without any modification as in row 610.

Figure 7 illustrates element select format for 8 8-bit elements in a 64-bit register. The exemplary vector register vt 700 is initially loaded with eight elements: A, B, C, D, E, F, G, and H. The present invention allows an instruction to select or specify any one of the element formats as indicated by rows 702 through 718. For example, element G for vt 700 may be selected and replicated for all 8 elements as shown in row 714. On the other hand, the vt 700 may be passed without any modification as in row 718.

ARITHMETIC INSTRUCTIONS

In accordance with the preferred embodiment of the present invention, arithmetic operations are performed on the corresponding elements of vector registers. The instruction is fetched from main memory and is loaded into a instruction register. It specifies the arithmetic operation to be performed.

In the following arithmetic instructions, the operands are values in integer vector format. The accumulator is in the corresponding accumulator vector format. The arithmetic operations are performed between elements of vectors occupying corresponding positions in the vector field in accordance with SIMD characteristics of the present invention. For example, an add operation between vs and vt actually describes eight parallel add operations between vs[0] and vt[0] to vs[7] and vt[7]. After an arithmetic operation has been performed but before the values are written into the accumulator, a wrapped arithmetic is performed such that overflows and underflows wrap around the Accumulator's representable range.

Accumulate Vector Add (ADDA.fmt). In the present invention ADDA.fmt instruction, the elements in vector registers vt and vs are added to those in the Accumulator. Specifically, the corresponding elements in vector registers vt and vs are added. Then, the elements of the sum are added to the corresponding elements in the accumulator. Any overflows or underflows in the elements wrap around the accumulator's representable range and then are written into the accumulator.

Figure 8 illustrates an exemplary ADDA.fmt arithmetic operation between elements of operand registers vs 800 and vt 802. Each of the registers 800, 802, and 804 contains 4 16-bit elements. Each letter in the elements (i.e., A, B, C, D, E, F, G, H, and I) stands for a binary number. FFFF is an hexadecimal representation of 16-bit binary number, 1111 1111 1111 1111. The vs register 800 holds elements FFFF, A, B, and C. The selected elements of vt registers are FFFF, D, E, and F. The ADDA.fmt arithmetic instruction directs the VU to add corresponding elements: FFFF+FFFF (=1FFFD), A+D, B+E, and C+F. Each of these sums are then added to the corresponding existing elements (i.e., FFFF, G, H, and I) in the accumulator 804: FFFF+1FFFD, A+D+G, B+E+H, and C+F+I. The addition of the hexadecimal numbers, 1FFFD and FFFF, produces 2FFFC, an overflow condition for a general purpose 64-bit register. The accumulator's representable range is 48 bits in accordance with the present invention. Since this is more than enough bits to represent the number, the entire number 2FFFC is written into the accumulator. As a result, no bits have been lost in the addition and accumulation process.

Load Vector Add (ADDL.fmt). According to the ADDL.fmt instruction, the corresponding elements in vectors vt and vs are added and then stored into corresponding elements in the accumulator. Any overflows or underflows in the elements wrap around the accumulator's representable range and then are written into the accumulator 706.

Figure 9 illustrates an exemplary ADDL.fmt arithmetic operation between elements of operand registers vs 900 and vt 902. Each of the registers 900, 902, and 904 contains 4 16-bit elements. Each letter in the elements (i.e., A, B, C, D, E, and F) stands for a binary number. FFFF is an hexadecimal representation of 16-bit binary number, 1111 1111 1111 1111. The vs register 900 holds elements FFFF, A, B, and C. The selected elements of vt registers are FFFF, D, E, and F. The ADDA.fmt arithmetic instruction instructs the VU to add corresponding elements: FFFF+FFFF , A+D, B+E, and C+F. The addition of hexadecimal numbers, FFFF and FFFF, produces 1FFFD, a technical overflow condition for a general purpose 64-bit register. The present invention wraps the number 1FFFD around the accumulator's representable range, which is 48 bits. Since this is more than enough bits to represent the number, the entire number 1FFFD is written into the accumulator. As a result, no bits have been lost in the addition process.

Accumulate Vector Multiply (MULA.fmt). The MULA.fmt instruction multiplies the values in vectors vt and vs. Then the product is added to the accumulator. Any overflows or underflows in the elements wrap around the accumulator's representable range and then are written into the accumulator.

Add Vector Multiply to Accumulator (MULL.fmt). The MULL.fmt instruction multiplies the values in vectors vt and vs. Then, the product is written to the accumulator. Any overflows or underflows in the elements wrap around the accumulator's representable range and then are written into the accumulator.

Subtract Vector Multiply from Accumulator (MULS.fmt). In MULS.fmt instruction, the values in vector vt are multiplied by the values in vector vs, and the product is subtracted from the accumulator. Any overflows or underflows in the elements wrap around the accumulator's representable range and then are written into the accumulator.

Load Negative Vector Multiply (MULSL.fmt). The MULSL.fmt instruction multiplies the values in vector vt with the values in vector vs. Then, the product is subtracted from the accumulator. Any overflows or underflows in the elements wrap around the accumulator's representable range and then are written into the accumulator.

Accumulate Vector Difference (SUBA.fmt). The present SUBA.fmt instruction computes the difference between vectors vt and vs. Then, it adds the difference to the value in the accumulator. Any overflows or underflows in the elements wrap around the accumulator's representable range and then are written into the accumulator.

Load Vector Difference (SUBL.fmt).. According to SUBL.fmt instruction, the differences of vectors vt and vs are written into those in the accumulator. Any overflows or underflows in the elements wrap around the accumulator's representable range and then are written into the accumulator.

ELEMENT TRANSFORMATION

After an arithmetic operation, the elements in the accumulator are transformed into the precision of the elements in the destination registers for further processing or for eventual storage into a memory unit. During the transformation process, the data in each accumulator element is packed to the precision of the destination operand. The present invention provides the following instruction method for such transformation.

Scale, Round and Clamp Accumulator (Rx.fmt). According to Rx.fmt instruction, the values in the accumulator are shifted right by the values specified in a vector field vt in the instruction opcode. This variable shift supports application or algorithm specific fixed point precision. The vt operands are values in integer vector format. The accumulator is in the corresponding accumulator vector format.

Then, each element in the accumulator is rounded according to a mode specified by the instruction. The preferred embodiment of the invention allows three rounding modes: 1) round toward zero, 2) round to nearest with exactly halfway rounding away from zero, and 3) round to nearest with exactly halfway rounding to even. These rounding modes minimize truncation errors during arithmetic process.

The elements are then clamped to either a signed or unsigned range of an exemplary destination vector register, vd. That is, the elements are saturated to the largest representable value for overflow and the smallest representable value for underflow. Hence, the clamping limits the resultant values to the minimum and maximum precision of the destination elements without overflow or underflow.

SAVING ACCUMULATOR STATE

Since the vector accumulator is a special register, the present invention allows the contents of the accumulator to be saved in a general register. However, because the size of the elements of the accumulator is larger than the elements of general purpose registers, the transfer occurs in multiple chunks of constituent elements. The following instructions allow storage of the accumulator state.

Read Accumulator (RAC.fmt). The RAC.fmt instruction reads a portion of the accumulator elements, preferably a third of the bits in elements, and saves the elements into a vector register. Specifically, this instruction method allows the least significant, middle significant, or most significant third of the bits of the accumulator elements to be assigned to a vector register such as vd. In this operation, the values extracted are not clamped. That is, the bits are simply copied into the elements of vector register, vd.

Write Accumulator High (WACH.fmt). The WACH.fmt instruction loads portions of the accumulator from a vector register. Specifically, this instruction method writes the most significant third of the bits of the accumulator elements from a vector register such as vs. The least significant two thirds of the bits of the accumulator are not affected by this operation.

Write Accumulator Low (WACL.fmt). According to WACL.fmt instruction, the present invention loads two thirds of the accumulator from two vector registers. Specifically, this instruction method writes the least significant two thirds of the bits of the accumulator elements. The remaining upper one third of the bits of the accumulator elements are written by the sign bits of the corresponding elements of a vector register such as vs, replicated by 16 or 8 times, depending on the data type format specified in the instruction.

A RACL/RACM/RACH instruction followed by WACL/WACH are used to save and restore the accumulator. This save/ restore function is format independent, either format can be used to save or restore accumulator values generated by either QH or OB operations. Data conversion need not occur. The mapping between element bits of the OB format accumulator and bits of the same accumulator interpreted in QH format is implementation specific, but consistent for each implementation.

The present invention, a method for providing extended precision in SIMD vector arithmetic operations, utilizes an accumulator register. While the present invention has been described in particular embodiments, it should be appreciated that the present invention should not be construed as being limited by such embodiments, but rather construed according to the claims below.